

CloneTracker: Advanced Dual-Layered Approach for Precise Detection of Clones in Java Applications

Yasir Glani*, Ping Luo, Syed Asad Shah, and Ke Lin

Abstract: In the fast-paced world of software development, conventional code clone detection methods are becoming inadequate, frequently struggling to identify complex clones. This gap significantly endangers software quality and security, emphasizing the need for advanced clone detection approach. In response, our CloneTracker introduces an advanced, dual-layered solution to identify all types of clones. CloneTracker utilizes a dual-layer detection strategy based on function-level and chunk-level analysis for enhanced accuracy. Specifically, it prioritizes function-based signature detection, followed by chunk-based detection, requiring at least 50% similarity and three matching chunks from the same file for clone identification. The dual-level indexing and synergistic abstraction distinguish our approach, surpassing traditional methods in efficiency and effectiveness. A comprehensive evaluation against well-known clone detection techniques across multiple reliable datasets revealed CloneTracker's peerless precision average of 98%, recall ranging from 99.0% to 100%, accuracy ranging from 97.5% to 99.0%, and efficiency of 1.14 to 1.29 s, with these results reflecting averages over function-based and file-based assessment. Its superior efficacy reaffirms the approach's importance in modern software development, setting a new standard for clone detection and paving the way for future advancements. Beyond mere detection, CloneTracker also precisely identifies changes in cloned code, enhancing code quality and maintenance practices in the modern software development paradigm.

Key words: code clone detection; software maintenance; software security; software reuse

1 Introduction

In the evolving field of software engineering, the practice of code cloning has become a focal point of interest, revealing the nuanced balance between the benefits and challenges of reusing code. This practice, commonly involving the copying and pasting of code blocks with minimal, significant, or no changes across

different parts of a software project, is a double-edged sword. While it can significantly accelerate the development process by allowing developers to reuse existing segments of code, it also incurs considerable maintenance challenges and the risk of propagating errors throughout the system. The identification of a bug in a single fragment of cloned code can lead to the complex task of implementing corrections across all occurrences where that code has been reused. This can lead to increased maintenance costs and inconsistencies, and a heightened risk of introducing bugs into the software system. This emphasizes the thorough identification and management of such clones within the source code.

Research findings indicate that cloned code constitutes approximately 10%–15% of the codebase in large software systems, highlighting its widespread

- Yasir Glani, Ping Luo, and Ke Lin are with Software School, Tsinghua University, Beijing 100084, China. E-mail: yasirglani@nwpu.edu.cn, yxe19@tsinghua.org.cn; luop@tsinghua.edu.cn; leonard.keilin@gmail.com.
- Syed Asad Shah is with Department of Computer Science, Bilkent University, Ankara 06800, Türkiye. E-mail: asad.shah@bilkent.edu.tr.

* To whom correspondence should be addressed.

Manuscript received: 2024-05-05; revised: 2024-09-01; accepted: 2025-03-10

use^[1]. For systems programmed in Common Business-Oriented Language (COBOL), the prevalence of cloned code is even higher, estimated at about 50%^[2]. This statistic not only highlights the prevalence of code cloning in software development but also emphasizes the urgent need for solutions that can adeptly navigate its benefits while minimising the risk of inefficiencies. Consequently, code cloning has become a focal point in discussions on enhancing software quality and maintainability in contemporary complex software projects^[3].

Figure 1 provides a comprehensive analysis of various code clone categories and types via a Java function designed for temperature conversion from Celsius to Fahrenheit. The original code segment delineates a basic function where temperatures input in Celsius are converted to Fahrenheit. This function also includes conditional logic based on the temperature, issuing a cold-weather alert when conditions are freezing and a dehydration warning during high temperatures. This code establishes the baseline for the direct clone category, which comprises Type-I and Type-II clone types involving modifying spaces, comments, and identifier name changes while maintaining essential logic.

The general clone category, represented by the Type-III Moderate-Type (MT) clone, introduces significant

logical enhancements to the original code. It retains the core conversion functionality while incorporating additional logic: A user prompt to ascertain whether a conversion from Fahrenheit back to Celsius is required. This enhancement not only improves the application's utility but also increases its interactivity, demonstrating the common developer practice of adapting and augmenting code to satisfy diverse and evolving user needs.

1.1 Background and motivation

The detection of code clones, especially Type-III Strong-Type (ST) and MT clones, poses significant challenges for traditional clone detection techniques. Type-III ST clones involve incremental modifications up to 30% new code, whereas Type-III MT or large-gap clones include more extensive modifications, such as altering up to 50% of the original code. Such extensive modifications, which can lead to notable structural and semantic changes, render these clones particularly elusive to detect using conventional methods.

Over recent decades, a variety of code clone detection techniques have been proposed, focusing primarily on identifying Type-I and Type-II clones and proposing simple text-based and token-based approaches. These have been effective for exact and

```
1 import java.util.Scanner;
2 // Converting celsius to fahrenheit
3 public class TempConverter {
4     public static void main(String[] args){
5         Scanner sc = new Scanner(System.in);
6         System.out.print("celsius: ");
7         double c = sc.nextDouble();
8         double f = (c * 9/5) + 32;
9         System.out.println("fahrenheit:"+f);
10        if (c <= 0){
11            System.out.println("freezing, wear more
12            clothes!");
13        } else if (c > 30) {
14            System.out.println("hot, stay hydrated!");
15        }
16    }
17 }
```

(a) Original code

```
1 import java.util.Scanner;
2 // Temp scales with checks
3 public class TempTransAdv {
4     public static void main(String[] args){
5         Scanner value = new Scanner(System.in);
6         System.out.print("input temp: ");
7         double tempC = value.nextDouble();
8         double tempF = (tempC* 9.0/5.0) + 32.0;
9         System.out.println("converted temp: " +
10         tempF);
11        // Advice based on temperature
12        if (tempC <= 0) {
13            System.out.println("freezing, stay inside
14            house!");
15        } else if (tempC >= 30) {
16            System.out.println("Hot, Drink more water
17            !");
18        }
19    }
20 }
```

(b) Direct clone (Type-I and Type-II)

```
1 import java.util.Scanner;
2 // Temp scales with checks
3 public class TempTransAdv {
4     public static void main(String[] args) {
5         Scanner value = new Scanner(System.in);
6         System.out.print("input celsius: ");
7         double tempC = value.nextDouble();
8         double tempF = (tempC* 9.0 / 5.0) + 32.0;
9         System.out.println("fahrenheit temp: " +
10         tempF);
11        if (tempC <= 0) {
12            System.out.println("freezing, stay
13            inside house!");
14        } else if (tempC >= 30) {
15            System.out.println("hot, drink more
16            water!");
17        }
18        // New code block: Optional conversion
19        System.out.print("convert from fahrenheit
20        to celsius? (yes/no): ");
21        String userResponse = value.next();
22        if (userResponse.equalsIgnoreCase("yes")) {
23            System.out.print("input Fahrenheit ");
24            tempF = value.nextDouble();
25            tempC = (tempF - 32)* 5.0 / 9.0;
26            System.out.println("Celsius temp: " +
27            tempC);
28        }
29    }
30 }
```

Upto 50% new code added

(c) General clone (Type III MT)

Fig. 1 Examples of Type-I, Type-II, and Type-III code clones.

syntactically similar clones but less so for Type-III ST and MT clones. Despite advances in Abstract Syntax Tree (AST)-based, Program Dependency Graph (PDG)-based, and hybrid detection methods, significant challenges remain in accurately identifying and pinpointing modifications made in these complex clone types.

Previously, function and chunk-based techniques separately have been proposed for code clone detection. While these techniques benefit from the efficiency gained through the use of indexing, they often suffer from lower precision, recall, F1-score, and accuracy due to the strict requirements for exact chunk signatures matching. In the function-based detection approach, the addition of even a single line of code can alter the entire signature of a function. Similarly, in chunk-based techniques, the large size of chunks and the strict adherence to exact whole-chunk matching signatures may undermine accuracy. Moreover, these techniques primarily identify potential clones rather than pinpointing specific modifications within the cloned code. To address these shortcomings and enhance the capability to not only detect clones but also accurately locate code modifications, we propose the CloneTracker approach. This innovative approach aims to fill the existing research gap by providing a more precise and efficient approach for clone detection and localization. In light of this, the objectives of this paper are addressed in Section 1.2.

1.2 Objectives of the paper

Efficient, effective, and precise clone detection is essential for maintaining code quality and managing complexity in software development. This research introduces CloneTracker, an innovative approach designed to outperform existing techniques by identifying complex and functionally similar code clones. The primary objectives of this study are outlined as below.

(1) Innovate: Introduce the dual-layered CloneTracker, a novel approach that integrates function-based detection with an advanced chunk signature matching system, enhanced with intelligent matching algorithms. This configuration not only enhances accuracy but also maintains high efficiency in detection time, effectively transforming the landscape of code clone detection by identifying a broad range of clone types, including Type-I, Type-II, and Type-III (ST, MT).

(2) Precise detection: Demonstrate CloneTracker's exceptional capability to not only detect clones effectively but also to accurately pinpoint and map specific modifications within clone pairs with high precision.

(3) Validate: Assess CloneTracker's effectiveness and superior performance in precision, recall, F1-score, and accuracy through extensive comparative evaluations against established techniques. This includes the ability to identify complex code clone (Type-III MT) modifications that traditional techniques may struggle to identify.

(4) Impact analysis: CloneTracker's role in enhancing software quality and optimizing maintenance workflows. Its profound insights into clone management underscore its crucial role in elevating code quality standards and establish CloneTracker as a groundbreaking approach in software engineering.

With these objectives, our research aims to revolutionize the standards of clone detection and equip developers and researchers with an essential approach to significantly improve the integrity and maintenance of software systems.

2 Literature Review

In recent decades, the search for more advanced code clone detection tools has accelerated, driven by the need to identify and locate the complex clones in extensive codebases more effectively. This endeavor aims to enhance software maintenance by improving the clone technique to precisely identify and locate clones in large-scale systems. Various innovative techniques have been proposed recently, each designed to address the complex challenges of detecting clones, which are summarized as follows.

Lexical-based techniques: Lexical/text-based comparison is a foundational technique in code clone detection, analyzing source code as plain text and comparing it line by line. This method has facilitated the identification of exact matches or clones through text-based similarity analyses. Research studies^[4, 5] have implemented text-based similarity analyses on source code sequences to identify exact matches. Innovations in this domain include Yu et al.'s multigranularity method^[6], which leverages Java bytecode for detecting up to Type-III clones by converting Java source into .txt format. Additionally, Chen et al.^[7] utilized NICAD for versatile Android

clone detection across various clone types. Lyu et al.'s SuiDroid^[8] employed layout XML files and the Context Triggered Piecewise Hashing (CTPH) hash algorithm for identifying Type-I, Type-II, and Type-III Android app clones. Despite these advances, traditional text-based methods have limitations due to exact text matching.

Token-based techniques: Various researchers^[9–11] have introduced token-based techniques for analyzing source code, converting it into a series of tokens processed as sequences to detect clones. Among these, Wang et al.^[12] proposed CCAaligner, employing C and Java files to detect Type-I, Type-II, and Type-III clones, leveraging the inherent structure of the code. Sajjani et al.^[13] proposed SourcererCC, using the IJA-Dataset and an inverted index data structure for quick querying of proportional clones, capable of identifying Type-I to Type-III clones. Yuki et al.^[14] presented a technique for detecting multi-grained code clones, utilizing Java files and the sequence alignment algorithm to identify identical hash sequences, effectively uncovering Type-I, Type-II, and Type-III clones.

Tree-based techniques: Tree-based techniques transform source code into an AST and employ tree-matching techniques to identify similar code. Yang et al.^[15] proposed a cloning technique that creates an AST from functions, transforms it into a new tree structure, and then applies the sequence alignment algorithm to score function similarity, detecting clones across various function levels in open-source projects. Chodarev et al.^[16] introduced an AST algorithm for pattern recognition to detect Type-I and Type-II clones. Pati et al.^[17] discussed a method leveraging AST and a Multi-Objective Genetic Algorithm (MOGA) for clone prediction across software, identifying Type-I, Type-II, and Type-III clones, and illustrating the diverse applications of tree-based analysis in clone detection.

PDG-based techniques: PDGs encapsulate control and data flow, providing essential semantic information for clone detection^[18, 19]. Upon obtaining PDGs, isomorphic subgraph matching algorithms are used to identify similar subgraphs as clones. Wang et al.^[20] introduced CCSharp, an efficient detector leveraging modified PDGs and Frama-C2 for generating these graphs. Sabi et al.^[21] investigated the impact of reordering Java code statements on detecting Type-I and Type-II clones. Crussell et al.^[22] developed a

technique to discover new clone relationships in Java code through PDG-based analysis.

Hybrid-based techniques: The hybrid method in code clone detection combines multiple methods such as Textual, Token, PDG, and AST^[23, 24]. Roopam and Singh^[25] proposed to enhance the code clone algorithm through a hybrid method that incorporates metric-based and PDG-based methods. This approach uses Java source code, converting it into AST and PDG formats to detect Type-I, Type-II, and Type-III clones. Misu and Sakib^[26] introduced an Interface Code Clone Detection (IDCCD) that combined token-based and metric-based approaches, utilizing the IJA-Dataset converted into regularized tokens and ASTs, capable of identifying Type-I to Type-III clones. Matsushita and Sasano^[27] introduced an algorithm that detects clones with large-gaps, leveraging a hybrid approach that combines token-based and AST-based methods. Uemura et al.^[28] proposed the Verilog HDL technique by combining token-based and metric-based techniques.

Following the introduction and review of literature review on code cloning, we outline preliminary definitions along with notations, proposed technique, evaluate its effectiveness, and conclude with discussions and summaries of our findings sections are as follow.

3 Preliminary Definition

This section introduces the fundamental concept of code cloning, as proposed by Saini et al.^[29]. We further outline our methodology by categorizing clones into two categories: The directed clone category, which includes Type-I and Type-II clones, and the general clone category, which includes Type-III ST and Type-III MT clones. These categories are essential for our detection approach, aiming to provide a nuanced understanding of clone types. This classification enhances the precision of our method in identifying and analyzing complex clones.

Code fragment: Code fragment is a sequence of statements, ranging from an entire function to a discrete block of code, outlined by specific numerical boundaries within a source file. Such fragments are formally identified by their source file S , the initial line number L_{start} , and the terminal line number L_{end} .

Code Similarity Ratio (SR): The SR measures the degree of cloning by comparing the statement count of

two code fragments (F_a) to another (F_b), represented as $SR_{a,b}$. It is important to recognize that $SR_{a,b}$ and $SR_{b,a}$ may have different statement lengths, reflecting the asymmetry in clone relationships.

Code clone/clone pair: Conceptually, a clone pair (C) involves two or more code sections (F_a, F_b) that are either identical or exhibit a high degree of similarity, potentially across different projects or within the same codebase. The pair's SR, which is the minimum of $SR_{a,b}$ and $SR_{b,a}$, provides a quantitative basis for clone identification.

Direct clone pair: A code fragment F_a contains some statements cloned from another code fragment, F_b , which can be described as a directed pair [$F_a, F_b, SR_{a,b}$]. In direct clones, the lengths of F_a fragment and F_b fragment are always the same, and modifications are only made on spaces, comments, and renaming variables, functions, etc. Type-I and Type-II clones are example of direct clone pair.

General clone pair: Code fragment F_a is copied code from another fragment F_b , which could be a function or a code block. The lengths of F_a and F_b do not need to be equal. Once F_a is copied from F_b , modifications can be made to F_a fragment through the insertion or deletion of code lines. In general clones, up to 30% of the code in F_a fragment is modified. This is also known as Type-III ST clone.

The classification of Type-III MT clones, while not universally defined, draws upon the foundational concepts of Higo et al.^[30]. This classification encapsulates scenarios where cloned code undergoes significant transformation, such as repeated insertions or deletions of statements or modifications affecting up to 50% of the original function's code over time. These extensive modifications categorize them as a specialized subset of general clones. An illustrative case of such extensive modification is shown in Fig. 1, highlighting a clone with substantial statement insertions. Despite their resemblance to Type-III ST clones, Type-III MT clones are distinguished by their scale of modification. Thus, they are considered separately within our analysis for their unique characteristics and implications.

4 Approach

This section introduces the step-by-step implementation of our proposed CloneTracker detection approach. Initially, reliable datasets are

collected and preprocessed, including phases such as data cleaning, normalization, and refinement. Following that, we implement a process of abstracting the code at different levels. Once the code is normalized and abstracted, we extract functions and simultaneously create overlapping chunks of the code. These functions and chunks are then transformed into unique signatures. We store these signatures in a database known as the signature vault, which is indexed for efficient retrieval. These stored signatures are essential for the later phase, where a diff algorithm is applied to identify similarities/dissimilarities, assisting in the identification of code clones and modifications in copied code. Upon detecting clones and changes, the system generates a detailed clone report.

Our technique spreads across several stages, functioning as a sequential pipeline where each stage relies on the output from the previous stage. Starting with downloading the raw source code, it undergoes a series of transformations detailed in Fig. 2. This process is adaptable to various Java datasets. The detailed steps of preprocessing are described as below.

4.1 Data preparation

This subsection describes the preprocessing processes used to optimize raw source code for efficient clone identification. This process consists of a series of steps, each aimed at improving the source code for future analyses. The steps performed during this preprocessing phase are outlined in the subsequent text.

4.1.1 Preprocessing

Preprocessing is essential in the CloneTracker technique, as it establishes the foundation for efficient and effective code clone detection and analysis. The preprocessing steps, as emphasized by Li et al.^[31], are essential for enhancing code quality and identification of clones. Preprocessing includes several essential steps designed to detect common modifications in source code as follows:

- **Data cleaning:** This step focuses on identifying and removing any anomalies or outliers that might generate noise.
- **Data reduction:** Aims to reduce the dataset, ensuring that the smaller set of data still provides reliable insights for analysis.
- **Data transformation:** Includes processes like normalization and discretization to standardize the data format.

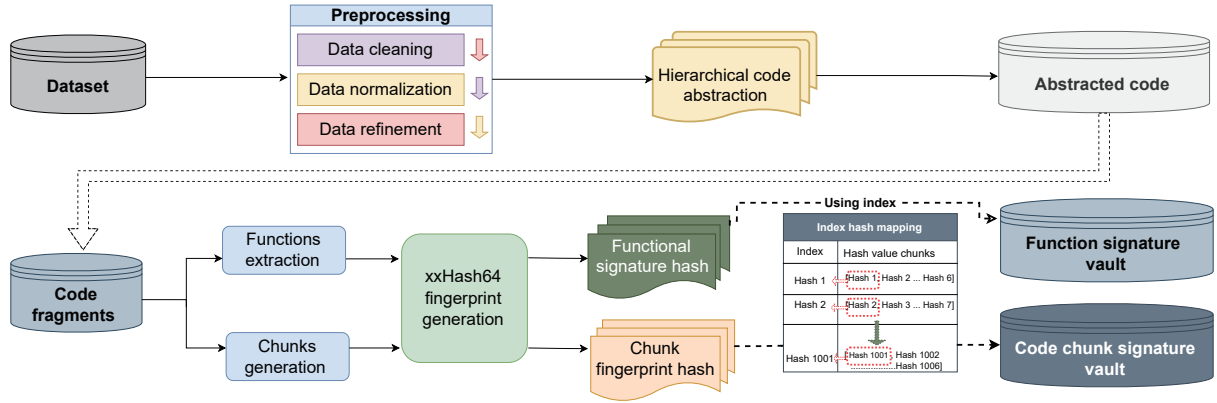


Fig. 2 Signature generation and database construction workflow in CloneTracker.

- **Data integration:** Merges data from different sources into a single database for unified processing.

These preprocessing principles significantly enhance the effectiveness of our clone detection technique. During the data cleaning phase, we remove lines of code that do not contribute to the functional behavior of the code but introduce noise, potentially affecting the accuracy of clone detection. This process is crucial for ensuring that our analysis focuses solely on meaningful code segments, thereby improving the precision of clone identification. The types of unnecessary lines of code removed are as below.

(1) **Package declarations and annotations**, which are typically used for metadata purposes rather than functional code behavior.

(2) **Comments**, both single-line and multi-line, which provide explanations or annotations but do not impact code execution.

(3) **White-spaces and empty lines**, which serve to improve code readability but do not affect its functionality.

By eliminating these unwanted lines of code, we prepare the source code into a form that is more facilitative to removing noise, thereby strengthening

the robustness and accuracy of our clone detection approach.

4.1.2 Synergistic abstraction

In our proposed CloneTracker approach, we address the challenges of using direct tokenization for cleaned source code in the clone detection techniques. Blind tokenization of cleaned code, whether at functions or overlapping chunks, misses true Type-II clones because copied code often goes through common modifications, such as renaming identifiers such as functions, variable name modifications, etc., and such blind tokenization practices should not be followed. To overcome this, we implement synergistic abstraction. Figure 3 shows an example of an abstraction process for calculating the area of a circle, which strengthens the CloneTracker approach to identify clones with common modifications. Synergistic abstractions are implemented as below.

- **Unified abstraction:** In one unified framework, the source code undergoes a thorough abstraction process. Classes, functions, strings, numeric literals, and variables are extracted, abstracted, and labeled with unique identifiers, ensuring accurate detection of code clones despite modifications. Classes and

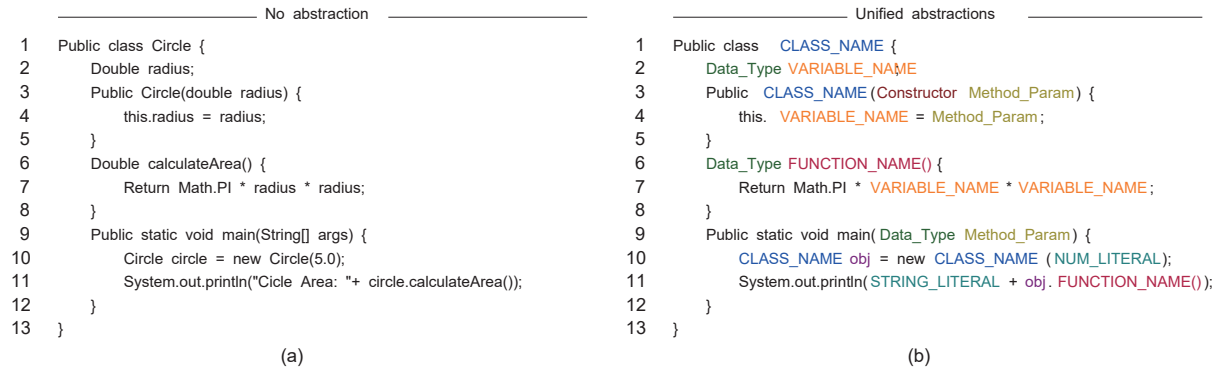


Fig. 3 Synergistic abstraction for code normalization. (a) Original source code; (b) unified abstracted code.

functions are tagged as “CLASS_NAME” or “FUNCTION_NAME”, while strings, numeric literals, and variables receive identifiers “STRING_LITERAL”, “NUM_LITERAL”, and “VARIABLE_NAME”, respectively. Additionally, constructors and method parameters are abstracted and stored in a list, labeled as “METHOD_PARAM”, to increase the approach’s resilience. This comprehensive abstraction process optimizes CloneTracker’s detection abilities, making it resilient to common code changes, ensuring effective identification of clones.

4.1.3 Token generation

Functions are retrieved from abstracted code to enhance the clone detection process. Recognizing that developers often reuse functions. Functions shorter than 5 lines are not extracted due to the implementation of unified abstraction; in a large database, this prevents the misidentification of non-clones as clones due to small size. If a small function is copied and reused, it will eventually be detected during the chunk’s signature-based detection. In parallel with extracting functions, the cleaned abstract code is split using semicolons to generate tokens. These tokens are then transformed into overlapping chunks using a sliding window size of 6; for instance, the first chunk contains 1–6 tokens, and the second chunk contains 2–7 tokens. This chunk generation and function extraction are essential to the CloneTracker approach. The chunk size significantly influences database creation time, detection speed, and the accuracy of clone identification. Oversized chunks, such as those containing 10 to 15 overlapping tokens, prolong preprocessing and database generation, impacting detection accuracy. In contrast, if the chunk’s size is too small, it accelerates the detection of false positive clone rates.

4.1.4 Signature generation

After functions and chunks are extracted and generated, they are transformed into signatures; traditionally, token-based code cloning methods used the MD5 hash algorithm to generate signatures. Due to multiple rounds of compression and large signature size, the MD5 hash algorithm is much slower to generate a large database, and it takes slightly longer to match clones in large databases due to the large size of signatures and chunk size. Previously, the FNV and Murmur hash algorithms have also been used. According to the hash algorithm benchmark, MD5, FNV, and Murmur algorithm efficiency is much slower than the xxHash64

hash algorithm, proposed by Westerdaal^[32]. Hash algorithms, which are and were traditionally used for code cloning, their benchmarks are as follows^[33]:

In Table 1, width represent the size of signature, and the quality represent the collision rate of the hash algorithm. We utilize the xxHash64 algorithm because it efficiently generates signatures at the near Random Access Memory (RAM) speed limit. Signature generation in clone detection should not only rely on the efficiency of the hash algorithm but also on the quality of the hash algorithm, which is measured in collision. The collision probability of xxHash is 1 in 10^{18} , meaning that the xxHash64 algorithm may generate an identical signature 1 in quintillion times. In other words, generating the same signature for two different tokens would be almost impossible in a more extensive database, making it a reliable signature generation algorithm for our proposed approach.

In the xxHash64 algorithm, as mentioned in Table 2, start with a seed value, which can be any 64-bit number. It converts the input string into the sequence of ASCII bytes, and following that, it processes the byte sequence in 32-byte chunks using xxHash64, involving several rounds of data mixing; following that, it updates the internal accumulators that capture the input data’s pattern through bitwise operations. Following that, it combines the internal states into a single 64-bit hash value, which can be either in decimal or hexadecimal. Signatures are generated in decimal format in our approach, which will be used for indexing.

4.1.5 Signature vault

Following the signatures generation for functions and overlapping chunks, which are subsequently stored in two specialized tables within a unified DataBase (DB) framework. The function signature vault table is structured with four columns as follows:

- (1) The entry number.
- (2) The function signatures in decimal format.
- (3) The abstracted code of the function.

Table 1 Hash function benchmark.

Hash name	Width	RAM speed (GB/s)	Hash quality score (1–10)
xxHash64	64-bit	19.4	10
Murmur3	32-bit	3.9	10
FNv64	64-bit	1.2	5
SHA1	160-bit	0.8	10
MD5	128-bit	0.6	10

Table 2 xxHash64 signature process.

Step	Process	Example ($a + b = c$)
1	Initialization	Seed is chosen (e.g., 0).
2	Byte conversion	String " $a + b = c$ " $\rightarrow$ byte array [97, 32, 43, 32, 98, 32, 61, 32, 99].
3	Hashing	The byte array is processed by the algorithm.
4	Accumulation	Accumulators capture the byte array's essence.
5	Finalization	A 64-bit hash is produced from internal states.
6	Output	Hash value is output in decimal.

(4) the file path from which the data is derived.

This design facilitates efficient retrieval of clone functions by indexing the decimal-formatted signatures in the second column. The code chunk signature vault table is organized into the following five columns, specifically designed for the storage of overlapping chunk signatures:

- (1) The entry number.
- (2) The initial (first) signature value of each chunk.
- (3) The overlapping chunk's signature.
- (4) The overlapping chunk's code.
- (5) The corresponding file path.

The indexing of the second column, which contains the first signature of each overlapping chunk, significantly enhances the efficiency of clone retrieval. This dual-table method within a single database not only streamlines the process of code clone detection but also optimizes the clone retrieval mechanisms, ensuring a robust and effective solution to identifying clones.

4.2 Clone detection process

In the clone detection process, the subject code undergoes a thorough preprocessing phase, including data cleaning, normalization, and data refinement. This phase is essential for cleaning and normalizing the code by removing unnecessary lines and preparing it for further analysis. Following this initial preprocessing, unified abstraction methods are implemented on the subject code. This involves the abstraction of functions, classes, string literals, numeric literals, variables, and constructor levels; the core structure of the subject code remains the same to ensure comprehensive analysis. After the abstraction, functions are systematically extracted from the subject code. Simultaneously, overlapping chunks of code are generated with a defined threshold of six lines.

xxHash64 signatures are generated against extracted functions and overlapping chunks. These chunks' signatures and extracted functions' signatures are then stored in memory, ready to be compared against the database vault for similarity checks.

4.2.1 Clone identification

In our clone detection framework as mentioned in Fig. 4, the process begins by comparing the signatures of functions extracted from the subject code against a pre-indexed functions database. This initial step efficiently identifies potential clones by leveraging the indexed structure for efficient matching. Function signatures are matched using Eq. (1),

$$SR_{\text{function}} = \begin{cases} 1, & \exists H(F_b) \in \text{DB such that } H(F_a) = H(F_b) \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The CloneTracker technique identifies direct clones, such as Type-I and Type-II, through implemented preprocessing and abstraction. If the hash signature of a function in the subject $H(F_a)$ exactly matches any function's hash signature in the database hash $H(F_b)$, is considered a potential clone.

If the subject's function's signature is not found within the database, our intelligent clone detection system will proceed with an advanced chunk signature matching method. This secondary phase is designed to efficiently identify more complex clones, including Type-III ST and Type-III MT clones that may not be directly identifiable through function signature alone. The intelligent matching system is structured so that if the function's signatures do not match with the database, it initiates the process of chunk signature matching. Instead of going through all entries in the database, this method takes the first signature from the subject's overlapping chunk and directly jumps to the line's signature hash value of the database (the first signature value of each chunk's signature in the database), which are also indexed and stored alongside chunk signatures. The chunk signature matching process operates under the premise that any significant portion of a signature, specifically 50% or more, must match between the subject code and the database. The computation algorithm for the chunk signature matching process is outlined as Eq. (2),

$$SR_{\text{chunk}} = \begin{cases} 1, & \exists 3 H(C_a) = H(C_b) \in \text{DB with match} \geq 50\% \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

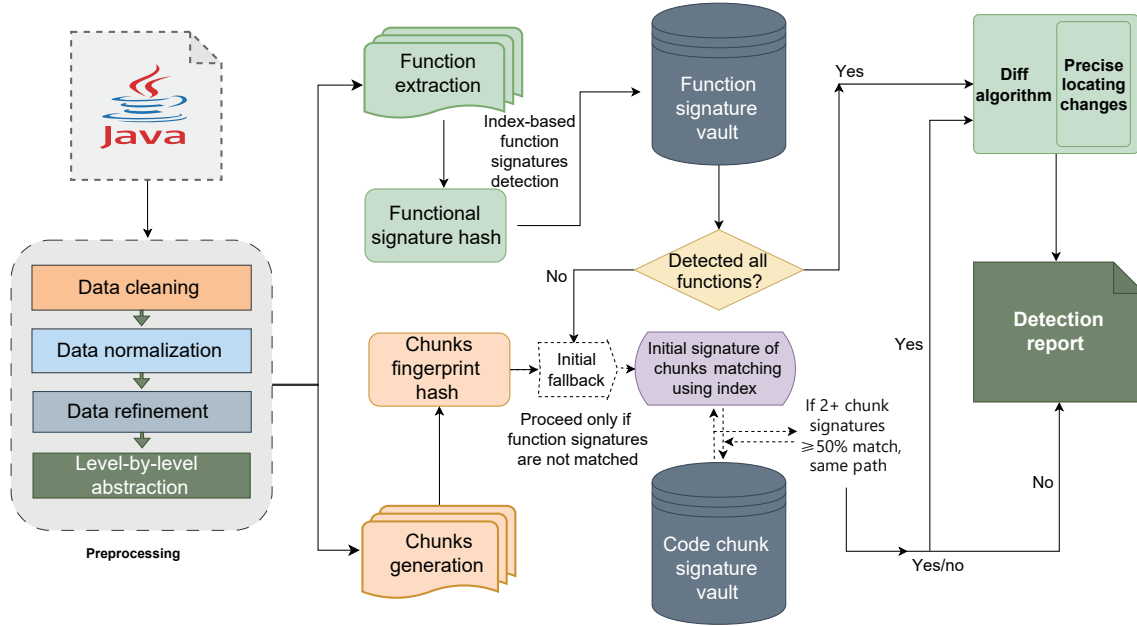


Fig. 4 Clone detection and localization workflow in CloneTracker.

To identify clone presence via chunk signature alignment, a minimum of three chunk correspondences within the same file, irrespective of their sequential or non-sequential arrangement, must meet the established matching threshold, which is 50% or more. Here, $H(C_a)$ denotes the chunk's hash signature from the subject code, while $H(C_b)$ refers to the hash signature within the database. This matching process of chunk does not follow any particular sequence, thereby granting greater adaptability in recognizing clones that might not be directly in sequence but still have a considerable degree of similarity. This approach ensures that our detection mechanism can accurately identify clones even when the code has undergone substantial modifications such as Type-III ST and Type-III MT clones.

4.2.2 Precise detection

Following the detection of code clones with an intelligent chunk signature matching algorithm, the diff algorithm is applied to identify potential changes within the copied code segments accurately. Using the diff algorithm marks a significant advancement in code cloning analysis. Since, complex clones have undergone extensive modifications, the diff algorithm is implemented to detect and identify the precise location of these modifications, pinpointing exact changes and providing a detailed view of modifications that are difficult to pinpoint by traditional methods. This method is efficient and effective for Type-III ST

clones and Type-III MT clones, as it facilitates modification discovery using the longest common subsequence within the code.

4.2.3 Algorithm diff

The diff algorithm plays an important role in identifying and analysing clones in CloneTracker, particularly in handling the complexities associated with Type-III ST and Type-III MT clones. This algorithm leverages a dynamic programming approach to determine the changes between two code segments. It begins the comparison of two sequences $X = [x_1, x_2, \dots, x_m]$ and $Y = [y_1, y_2, \dots, y_n]$, where X is the subject code, and Y is the code stored in the database is the database file.

To compute the LCS length between sequences X and Y , the algorithm performs the following steps:

• Step 1: Initialize the table.

- (1) Create table L of size $(m+1) \times (n+1)$.
- (2) $L[i][j]$ stores the length of the LCS of $X[1, 2, \dots, i]$ and $Y[1, 2, \dots, j]$.
- (3) Set all $L[i][j]$ to 0, indicating no common subsequence found yet.

The algorithm starts by creating a table L , which is of size $(m+1) \times (n+1)$ where m and n are the lengths of sequences X and Y , respectively. This table stores the lengths of the Longest Common Subsequence (LCS) between all prefixes of X and Y . Initially, all cells in the table are set to zero, representing that, at the beginning, no common subsequence has been identified. After initializing the table, the algorithm

proceeds to populate it using the LCS recurrence relations.

• **Step 2: Filling the table.**

- (1) Fills table L using recurrence relations.
- (2) If $x_i = y_j$, then $L[i][j] = L[i-1][j-1] + 1$.
- (3) Else, $L[i][j] = \max(L[i-1][j], L[i][j-1])$.

The table is then filled based on recurrence relations that compare each character of X with each character of Y . If a match is found (i.e., if $x_i = y_j$), the algorithm concludes that a common subsequence has been extended by one character. Thus, it increments the value found in the cell diagonally up and left from the current cell ($L[i-1][j-1]$) by 1 and stores it in $L[i][j]$. If x_i and y_j do not match, the algorithm takes the maximum value from either directly above ($L[i][j-1]$) or directly to the left ($L[i-1][j]$) and assigns it to $L[i][j]$. This step ensures that the length of the LCS up to that point is carried forward.

• **Step 3. Traceback for LCS:** Once the table is completely filled, the LCS can be determined by a traceback method starting from $L[m][n]$ (the bottom-right cell of the table) and moving backwards through the table, following the rules that led to the filling of the table.

This involves checking if the current cell's value was obtained from the cell above to the left or diagonally up and left. The traceback path indicates the sequence of matches and mismatches.

• **Step 4. Extracting the LCS:** During the traceback, when the algorithm moves diagonally, it indicates that a matching character has been found, and that character is part of the LCS.

By following the traceback path and recording these characters, the LCS is extracted, representing the longest subsequence of code present in both the subject file and the database file.

For practicality in comparing extensive source code, we consider a subject Java source file X , denoted by the sequence (A, B, C, D, E, F) , with each letter symbolizing a line of Java code. Similarly, Y is a database source file, which has (A, B, C, D, H, I, F) as its element sequence, where each letter corresponds to a line of Java code identified by our technique as a potential clone. The diff algorithm begins the comparison at the first identical line in both files, extending the analysis to the last matching line. A comparison table is created, with dimensions $(m+1) \times (n+1)$ to accommodate the lengths of X and Y (Step 1 in Fig. 5). This table's initial row and column

are set to zero, signifying the onset of the sequence comparison (Step 2 in Fig. 5). As X is compared against Y , table L is filled: Matching characters cause an increment by one from the diagonally preceding cell, while mismatches take the higher value from the adjacent left or upper cell (Step 3 in Fig. 5). Upon filling the LCS matrix, the traceback process is initiated from the last cell, moving upwards diagonally when a match is highlighted in blue. In case of a mismatch, the process selects the higher value from the adjacent top or left cell, which is highlighted in red and continues tracing back diagonally (Step 4 in Fig. 5). This process effectively identifies the identical lines and modifications in the subject source code file. This step is essential for reconstructing the longest common subsequence and pinpointing exact code matches and modifications.

5 Evaluation

In this evaluation section, we present a comprehensive comparison of our hybrid-token code cloning detection approach, CloneTracker against eight other known techniques. These techniques mainly include token-based and hybrid token-based methods, namely VCIPR^[34], VUDDY^[35], ICDT^[36], and AYAT^[37], which align closely with our hybrid token-based approaches. To highlight CloneTracker's versatility and provide a comprehensive assessment, we broadened our analysis to encompass various methods, such as hybrid lexical and metric-based method LWH^[38], and hybrid-PDG method, proposed by Song et al.^[39]. We also analyse and compare a method based on the AST by Yang et al.^[40], and a novel proposal by Liu et al.^[41] that employs sequence alignment, a method broadly used in bioinformatics, unique and different from traditional cloning detection methods like lexical-based, token-based, PDG-based, AST-based, or hybrid-based methods. This comprehensive comparative analysis emphasizes CloneTracker's robustness across different cloning detection methods. For techniques without designated tool names, we reference them by their author names for clarity in our graphical and tabular comparisons. Our assessment aims to measure each technique's ability to detect a range of clone types, particularly Type-I, Type-II, Type-III ST, and Type-III MT clones.

This study further investigates how various cloning detection techniques perform in scenarios mirroring actual development practices, especially when

Step 1: Table with sequence labels								
		A	B	C	D	H	I	F
A								
B								
C								
D								
E								
F								

Step 2: Initialize with zeros								
		A	B	C	D	H	I	F
	0	0	0	0	0	0	0	0
A	0							
B	0							
C	0							
D	0							
E	0							
F	0							

Step 3: LCS table filled								
		A	B	C	D	H	I	F
	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
B	0	1	2	2	2	2	2	2
C	0	1	2	3	3	3	3	3
D	0	1	2	3	4	4	4	4
E	0	1	2	3	4	4	4	4
F	0	1	2	3	4	4	4	5

Step 4: Traceback								
		A	B	C	D	H	I	F
	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
B	0	1	2	2	2	2	2	2
C	0	1	2	3	3	3	3	3
D	0	1	2	3	4	4	4	4
E	0	1	2	3	4	4	4	4
F	0	1	2	3	4	4	4	5

Fig. 5 Identifying modifications in cloned code using the diff algorithm.

developers copy specific functions or code snippets from existing codebases, and reuse entire functions, snippets, or source files with or without modifications to meet their required tasks. Our primary aim is to present a thorough comparison of CloneTracker against known cloning detection techniques, focusing on its practical application and effectiveness in real-world cloning scenarios.

5.1 Clone parameter settings

In the realm of code detection research, there is no universally accepted standard for defining the clones. However, the BigCloneBench and BigCloneEval benchmarks, developed by Jeffery Svajlenko and Roy^[42, 43] and based on the IJA-Dataset^[44], suggest sequences of six copied lines as a clone. This standard is widely adopted by several state-of-the-art code cloning detection techniques. Contrary to this, some researchers propose different criteria for identifying clones, including function, block, statement-level, and file-level clones.

Various techniques have their own definitions. For instance, the SQVDT^[45] identifies a clone as any consecutive 15-line sequence that is copied, VUDDY^[35] defines a copied function as a clone with

or without common code modifications, and the AYAT^[37] technique considers five consecutive copied lines as a clone. Liu et al.'s technique^[41] defines a clone based on half of the copied source code in a file, while Yang et al.'s technique^[40] proposes that if 90% of the nodes in the function (using AST-based) are similar, it will be known as a clone. Our approach, CloneTracker, defines clones based on overlapping chunks of six lines with a 50% matching ratio or more within the same file, where at least three chunks may be either consecutive or non-consecutive. That makes our approach also follow the BigCloneBench definition of clone, which is widely adopted.

5.2 Dataset

To assess our code clone detection approach, we utilize three well-known and reliable datasets: IJA-Dataset^[44], Github Java Corpus^[46], and Data for Java (D4J-Dataset)^[47]. The Github Java Corpus dataset is abbreviated as GTH-Dataset in our research study. We select a diverse subset of 100 000 Java files from both the IJA-Dataset and GTH-Dataset for experimental analysis, while opting to utilize the entirety of the D4J-Dataset for a comprehensive analysis.

Our preprocessing steps are implemented in

accordance with the steps specified in Section 4.1. Recognizing the distinct preprocessing requirements of other clone detection techniques, we adapte their preprocessing steps. This adjustment enabled an effective and fair comparison across different clone detection methodologies.

5.3 Experimental setup

To evaluate our code clone detection approach, we creat a testing environment with 200 Java files from each of the D4J-Dataset, IJA-Dataset, and GTH-Dataset, all initially categorized as non-clones. We systematically organize twelve groups for function-level analysis to cover each clone Type-I, Type-II, Type-III ST, and Type-III MT across the datasets. For each clone type, we systematically select one or two functions F_b from F_a in 100 files from the respective dataset and applied modifications through a custom script. This is done to ensure a diverse range of modifications suitable for each clone type. The objective is to maintain a 100% similarity ratio ($SR_{a,b}$) for direct clone categories, integrating permissible modifications that ranged from simple whitespace and comment adjustments in Type-I clones to more complex changes in variable names, function names, numeric and string literals, as well as structural modifications for Type-II clones. These modifications are strictly applied according to Eq. (3),

$$DC(F_a, F_b) = \begin{cases} 1, & SR(F_a, F_b) = 100\% \text{ permissible} \\ & \text{modifications (Type-I/Type-II)} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

For the general category Type-III ST clone, we systematically add up to 30% new code to the copied functions, in addition to the modifications of Type-I and Type-II clones. The criteria for generating Type-III ST clones are defined as Eq. (4),

$$\begin{aligned} & \text{Type-III ST}(F_a, F_b) \\ & = \begin{cases} 1, & 1.3 \leq \frac{\text{length of cloned code}(F_b)}{\text{length of original code}(F_a)} \text{ includes} \\ & \text{modifications (Type-I/Type-II)} \\ 0, & \text{otherwise} \end{cases} \end{aligned} \quad (4)$$

For the Type-III MT clone, the cloning process allows for significant modifications, with 30%–50% new code added at various points to the copied functions, in addition to the Type-I and Type-II clones. The criteria for generating Type-III MT clones are defined as Eq. (5),

$$\begin{aligned} & \text{Type-III MT}(F_a, F_b) \\ & = \begin{cases} 1, & 1.3 \leq \frac{\text{length of cloned code}(F_b)}{\text{length of original code}(F_a)} \leq 1.5 \text{ includes} \\ & \text{modifications (Type-I/Type-II)} \\ 0, & \text{otherwise} \end{cases} \end{aligned} \quad (5)$$

Following that, modified clone functions are then distributed systematically into 100 out of the 200 original non-clone files for each clone type of each dataset, effectively transforming them into clone files. The clone files are renamed “Clone-” followed by their original name, such as “Clone-Example.java”. Meanwhile, the files that did not undergo this process of clone code insertion remained classified as non-clones.

In parallel to the function-based analysis, we also structure 12 additional groups for file-level clone detection, similar to the function-based clone detection environment, resulting in a total of 24 groups. For file-based 12 groups, we directly inject Type-I, Type-II, Type-III ST, and Type-III MT clone modifications following Eqs. (3)–(5) to 100 selected files taken from respective dataset, transforming 100 files to known clone files, while keeping 100 files as non-clone files, where the size of these files ranged from small to large, containing up to 300 lines of code. Specifically, for Type-III MT modifications, the length of these files’ code is increased by 30%–50%. For example, a file originally containing 300 Lines Of Code (LOC) becomes 450 LOC.

This approach aims to imitate real-world scenarios, where developers might copy a function or entire files, and reuse them either with or without modifications. Implementing a systematic strategy across all datasets enabled a thorough and fair evaluation of our clone detection technique’s capabilities, effectively simulating the practical challenges of identifying both function-level and file-level clones in actual software development environments. This testing framework comprehensively covers the detection of Type-I, Type-II, Type-III (including ST and MT), ensuring a robust evaluation of our clone detection capabilities across all clone types.

5.4 Evaluation metrics for function-level clones

This section delves into the evaluation of function-level clone detection, highlighting the comparative analysis of our CloneTracker against advanced clone detection techniques. The emphasis is on accurately identifying

Type-I, Type-II, Type-III ST, and Type-III MT clones. For a comprehensive assessment, we employ evaluation metrics such as precision, recall, F1-score, accuracy, and detection time. The comprehensive results of function-base clone detection are mentioned in Tables 3 and 4. Average function-based clone results over all datasets are presented in Fig. 6. The following subsections (Sections 5.4.1–5.4.5) offer an in-depth analysis of these metrics, shedding light on the effectiveness and efficiency of each cloning detection technique in the code cloning area.

5.4.1 Precision

Precision measures the accuracy of a clone detection technique in correctly identifying actual clones. It calculates the proportion of identified clones that are true clones by dividing the number of true positives (TP) detections by the total number of detected clones,

which includes both TP and false positives (FP). Precision is computed using Eq. (6),

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (6)$$

Our approach demonstrates outstanding performance in code clone detection across all datasets. It achieves average precision rates of 98.3% for Type-I and Type-II clones, 98% for Type-III ST clones, and 98% for Type-III MT clones across all datasets, including IJA-Dataset, D4J-Dataset, and GTH-Dataset. This impressive precision emphasizes its superior ability to identify true clones. In contrast, VUDDY^[35] shows an average precision of 54% for all clone types.

Song et al.'s technique^[39] and Liu et al.'s technique^[41] achieve moderate average precision rates, around 54%–55% and 51%–67%, respectively. Yang et al.'s technique^[40] and LWH^[38] demonstrate slightly

Table 3 Function-based direct clones evaluation metrics. D.Time stands for detection time.

Technique	Dataset	Type-I					Type-II				
		Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)
Our technique	D4J-Dataset	100	100	100	100	1.07	100	100	100	100	1.06
	IJA-Dataset	98	100	99	99	1.63	98	100	99	99	1.61
	GTH-Dataset	97	100	98	98	0.76	97	100	98	98	0.76
VUDDY ^[35]	D4J-Dataset	67	91	77	73	0.42	66	89	76	72	0.27
	IJA-Dataset	45	45	45	45	0.37	45	46	46	45	0.27
	GTH-Dataset	52	59	55	53	0.40	52	59	55	52	0.25
Song et al.'s technique ^[39]	D4J-Dataset	65	93	77	71	6.27	65	94	77	72	3.98
	IJA-Dataset	45	46	46	45	6.72	46	47	46	46	4.91
	GTH-Dataset	55	67	61	56	5.87	52	60	56	53	4.53
Yang et al.'s technique ^[40]	D4J-Dataset	46	63	53	44	1.18	44	58	50	41	1.14
	IJA-Dataset	47	50	49	47	1.65	48	51	49	47	1.59
	GTH-Dataset	47	65	55	47	0.85	49	69	57	48	0.89
VCIPR ^[34]	D4J-Dataset	62	8	14	52	2.44	64	9	16	52	0.39
	IJA-Dataset	38	5	9	48	1.91	43	6	11	49	0.31
	GTH-Dataset	92	44	59	70	2.06	88	28	42	62	0.35
ICDT ^[36]	D4J-Dataset	53	57	55	54	3.00	51	52	51	51	1.47
	IJA-Dataset	84	53	65	71	0.54	81	43	56	67	0.18
	GTH-Dataset	60	64	62	61	2.73	59	58	58	53	1.74
AYAT ^[37]	D4J-Dataset	83	86	84	84	3.62	83	90	87	86	2.84
	IJA-Dataset	78	79	79	79	3.52	61	35	45	56	3.62
	GTH-Dataset	81	100	90	89	2.10	81	100	90	89	1.53
Liu et al.'s technique ^[41]	D4J-Dataset	58	97	72	63	1181.40	50	72	59	51	3368.98
	IJA-Dataset	50	1	2	5	23 049.74	50	1	2	5	24 049.10
	GTH-Dataset	54	100	70	59	7302.00	54	100	70	59	7379.00
LWH ^[38]	D4J-Dataset	65	64	65	65	2.94	65	64	65	65	4.33
	IJA-Dataset	21	17	19	27	6.81	20	16	18	27	2.35
	GTH-Dataset	53	57	55	54	3.16	53	58	56	54	3.36

Table 4 Function-based general clones evaluation metrics. D.Time stands for detection time.

Technique	Dataset	Type-III ST					Type-III-MT				
		Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)
Our technique	D4J-Dataset	100	100	100	100	1.08	100	100	100	100	1.10
	IJA-Dataset	98	100	99	99	1.63	98	100	99	99	1.69
	GTH-Dataset	96	91	93	94	0.76	96	85	90	91	0.83
VUDDY ^[35]	D4J-Dataset	66	86	74	70	0.29	65	83	73	69	0.29
	IJA-Dataset	47	49	48	47	0.29	46	48	47	46	0.30
	GTH-Dataset	51	58	54	52	0.27	51	57	54	51	0.29
Song et al.'s technique ^[39]	D4J-Dataset	65	92	76	71	3.91	64	89	74	69	3.52
	IJA-Dataset	47	50	49	47	5.20	47	49	48	47	4.80
	GTH-Dataset	52	59	55	52	6.78	51	57	54	51	7.85
Yang et al.'s technique ^[40]	D4J-Dataset	43	57	49	41	1.23	44	58	50	41	1.17
	IJA-Dataset	48	51	49	47	1.60	46	48	47	46	1.58
	GTH-Dataset	48	66	55	47	0.87	47	63	54	46	0.84
VCIPR ^[34]	D4J-Dataset	62	8	14	52	0.39	64	9	16	52	0.38
	IJA-Dataset	33	4	7	48	0.33	43	6	11	49	0.41
	GTH-Dataset	85	23	36	59	0.34	85	22	35	59	0.32
ICDT ^[36]	D4J-Dataset	53	56	54	53	1.46	54	59	56	55	1.29
	IJA-Dataset	81	43	56	67	0.24	85	55	67	72	0.24
	GTH-Dataset	53	47	50	53	1.36	53	46	49	53	1.22
AYAT ^[37]	D4J-Dataset	83	86	84	84	2.90	83	89	86	85	2.99
	IJA-Dataset	79	84	82	81	9.21	77	74	76	76	12.33
	GTH-Dataset	81	95	87	86	1.54	81	95	87	86	1.55
Liu et al.'s technique ^[41]	D4J-Dataset	49	68	57	49	3368.08	47	64	54	47	4365.30
	IJA-Dataset	50	1	2	5	25 186.92	100	100	99	99	27 814.00
	GTH-Dataset	54	100	70	59	7496.40	54	100	70	59	7797.00
LWH ^[38]	D4J-Dataset	53	87	66	56	6.21	56	95	70	59	4.88
	IJA-Dataset	55	92	69	58	3.82	50	76	60	50	4.18
	GTH-Dataset	56	87	68	59	5.86	56	87	68	59	3.36

lower precision rates, averaging around 46%–47% and 46%–54%, respectively. VCIPR^[34] achieves an average precision of 60% to 65% across all types of clones. ICDT^[36] demonstrates balanced performance, with precision rates ranging from 62% to 65%. However, AYAT^[37] impresses with a higher average precision, ranging from 75% to 80% across all clone types. While AYAT^[37] demonstrates its effectiveness in identifying clones, its precision remains significantly lower than that of our technique.

5.4.2 Recall

Precision and recall both are essential metrics for evaluating the efficacy of clone detection techniques. Specifically, recall is indicative of a technique's ability to accurately identify all clones in a dataset. This metric reflects the detection technique's proficiency in identifying a significant proportion of true code clones, where a higher recall value suggests a superior ability

of the technique to detect the actual clones present in the dataset. Recall is computed using Eq. (7), whereas FN denotes the number of false negatives.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (7)$$

Our approach demonstrates outstanding performance in code clone detection across all datasets, achieving average recall rates of 100% for Type-I clones, 98.3% for Type-II clones, 98% for Type-III ST clones, and 95% for Type-III MT clones. This high recall highlights its exceptional ability to detect true clones across all datasets, including IJA-Dataset, D4J-Dataset, and GTH-Dataset. VUDDY^[35] shows recall rates ranging from 62% to 65%, struggling particularly with more complex clones.

Song et al.'s technique^[39] and Liu et al.'s technique^[41] show moderate average recall rates, ranging from 65% to 68% for Song et al.'s

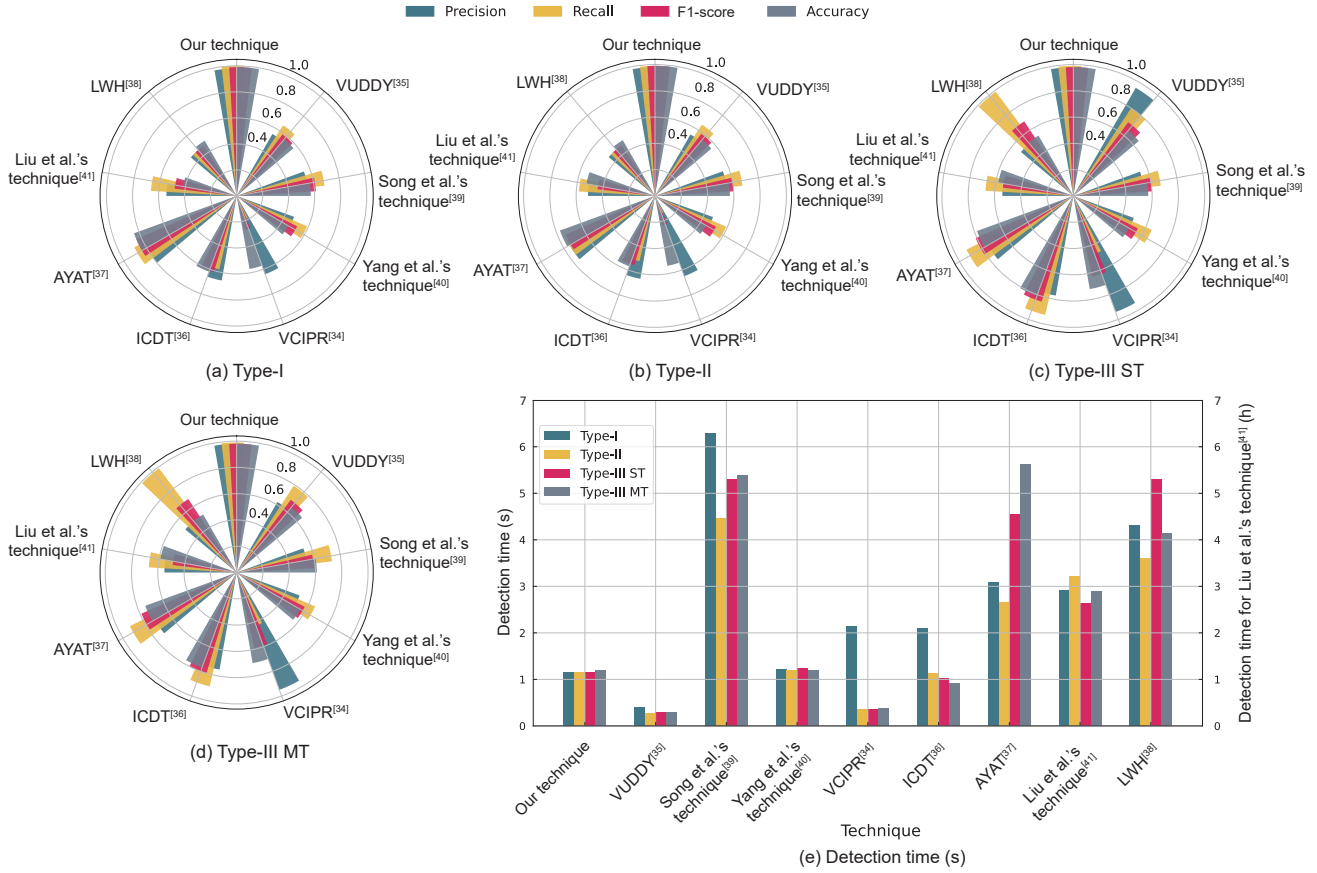


Fig. 6 Average function-based clone evaluation metrics over all datasets.

technique^[39] and 56% to 88% for Liu et al.'s technique^[41]. Yang et al.'s technique^[40] and LWH^[38] achieved recall rates between 56% and 59% and 46% to 88%, respectively. VCIPR^[34] attains below average recall rates from 11% to 19% across all types of clones, which is the lowest among all techniques. ICDT^[36] demonstrates balanced performance, with average recall rates ranging from 48% to 58%. AYAT^[37] shows higher recall rates, ranging from 75% to 81% across all clone types, demonstrating its effectiveness in identifying clones, though its recall remains lower than our technique.

5.4.3 F1-score

F1-score is an essential metric for evaluating code clone detection techniques, combining precision and recall to measure a technique's balanced performance. It captures the critical balance between detecting true clones and minimizing the false identification of clones. F1-score is determined by the harmonic mean of precision and recall, and it is computed as Eq. (8),

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (8)$$

This equation adeptly merges the precision and recall metrics, providing a comprehensive measure of a clone detection technique's accuracy and effectiveness. F1-score demonstrates the overall effectiveness of clone detection techniques. Our approach demonstrates outstanding performance in code clone detection, achieving average F1-scores of 99% for Type-I and Type-II clones, 97.33% for Type-III ST clones, and 96.33% for Type-III MT clones across the IJA-Dataset, D4J-Dataset, and GTH-Dataset. In contrast, VUDDY^[35] shows an average F1-score of 59% across all datasets. Song et al.'s technique^[39] and Liu et al.'s technique^[41] achieve around 61%–60% and 48%–74%, respectively.

Yang et al.'s technique^[40] and LWH^[38] lag behind with average F1-scores of 52%–67% and 46%–67%, respectively. VCIPR^[34] demonstrates an F1-score of 27% for Type-I clones but drops significantly to 19% for Type-III and 21% for Type-III MT clones. ICDT^[36] demonstrates balanced performance, with average F1-scores ranging from 53% to 61%. AYAT^[37] impresses with an average F1-score of 84% for Type-I and Type-

III ST clones, and 83% for Type-III MT clones, showcasing its proficiency in specific clone detection. However, AYAT^[37] lags behind CloneTracker, particularly in Type-II clones.

5.4.4 Accuracy

Accuracy is an essential metric for assessing the effectiveness of clone detection techniques, reflecting the rate at which cloned and non-cloned code segments are correctly identified. This measure takes into both TP and account for true negatives (TN), as well as FP and FN, to provide a comprehensive performance of code clone techniques. The accuracy is computed using Eq. (9),

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (9)$$

Our approach leads with exceptional accuracy, achieving averages of 99% for Type-I and Type-II clones, 97.67% for Type-III ST clones, and 96.67% for Type-III MT clones. This consistent performance underscores its precision in distinguishing both cloned and non-cloned code. Meanwhile, VUDDY^[35] achieves an average accuracy of 58%–59% across all clone types, reflecting variability in its detection efficiency.

Song et al.'s technique^[39] and Liu et al.'s technique^[41] achieve average accuracies of 55%–57% and 37%–68%, respectively. Yang et al.'s technique^[40] and LWH^[38] show average accuracies of 44%–47% and 49%–57%, respectively. VCIPR^[34] exhibits balanced detection across all clone types. ICDT^[36] demonstrates steady performance, with average accuracy ranging from 57% to 62%. AYAT^[37] excels, particularly in Type-I clones, with an impressive average accuracy of 84%, along with 84% and 82% strong performance in Type-III ST and Type-III MT clones. However, it lags behind in Type-II clones, achieving an average accuracy around 77%, due to not implementing any abstraction in the proposed technique.

5.4.5 Efficiency (detection time)

In the domain of code clone detection, the efficiency of detection notably emphasizes the effectiveness of clone detection techniques. Our proposed technique achieved excellent efficiency, averaging 1.15 s for Type-I, 1.14 s for Type-II, 1.15 s for Type-III ST, and 1.20 s for Type-III MT clones across the IJA-Dataset, D4J-Dataset, and GTH-Dataset. This excellent efficiency is achieved using an intelligent indexing method as mentioned in Section 4.1.5. The efficient matching algorithm is essential for processing and matching

extensive databases efficiently, highlighting its robustness in operational scenarios.

Other techniques demonstrate diverse efficiency levels. Average detection times of VUDDY^[35] remain around 0.29–0.39 s for Type-I–Type-III MT clones, VUDDY^[35] achieves incredible efficiency due to purely function-based technique, which matches signature using index, but it suffers from lower precision, recall, and accuracy especially for Type-III ST and Type-III MT. On the other hand, Song et al.'s technique^[39] takes an average detection time of 6.2 s for Type-I clones, 4.47 s for Type-II, 5.2 s for Type-III ST, and 5.39 s for Type-III MT. Yang et al.'s technique^[40] and AYAT^[37] show longer average detection times, notably from 2.6 to 5.6 s. Remarkably, Liu et al.'s technique^[41] technique, employing the complex Smith-Waterman algorithm for every subject's signature matching with the database, experiences significantly prolonged detection times, averaging over 10 500 s. This extensive duration is attributed to the algorithm's complex matrix computations, particularly when handling extensive databases.

5.5 Evaluation metrics for file-level clones

This section evaluates file-level clone detection, comparing our CloneTracker approach against state-of-the-art clone detection techniques. With a keen emphasis on accurately identifying Type-I, Type-II, Type-III ST, and Type-III MT clones, we utilize a comprehensive set of metrics — precision, recall, F1-score, accuracy, and detection time, allowing for a nuanced understanding of each technique's performance in the file-level clone detection context. Detailed results of file-based clone detection are presented in Tables 5 and 6, while the average results across all datasets are shown in Fig. 7. These provide insights into the efficiency and effectiveness of each clone detection technique.

5.5.1 Precision

In evaluating file-level clone detection precision, our proposed CloneTracker approach outperforms other techniques, achieving an average of 98% across Type-I, Type-II, Type-III ST, and Type-III MT clones across all datasets. This performance highlights its consistency and reliability in accurately identifying all clone types across all datasets. VCIPR^[34], demonstrate its strengths, achieving an average precision of 95% in Type-III MT clones, closely approaching our

Table 5 Evaluation metrics for file-based direct clones. D.Time denotes detection time.

Technique	Dataset	Code clone Type-I					Code clone Type-II				
		Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)
Our technique	D4J-Dataset	100	100	100	100	0.89	100	100	100	100	0.89
	IJA-Dataset	98	100	99	99	1.98	98	100	99	99	1.98
	GTH-Dataset	97	98	97	97	0.82	96	98	97	97	0.86
VUDDY ^[35]	D4J-Dataset	68	97	80	76	0.28	67	93	78	74	0.26
	IJA-Dataset	61	86	72	66	0.41	60	75	66	62	0.36
	GTH-Dataset	60	81	69	64	0.24	59	72	65	60	0.23
Song et al.'s technique ^[39]	D4J-Dataset	67	100	80	75	4.08	66	97	79	73	4.63
	IJA-Dataset	64	99	75	71	7.25	60	82	69	64	7.05
	GTH-Dataset	63	92	74	69	8.92	60	81	69	64	8.89
Yang et al.'s technique ^[40]	D4J-Dataset	53	86	66	56	1.09	51	80	62	52	1.10
	IJA-Dataset	58	77	66	61	1.74	54	67	60	55	1.66
	GTH-Dataset	54	85	66	56	0.91	52	75	63	54	0.87
VCIPR ^[34]	D4J-Dataset	95	100	95	97	1.55	96	52	68	75	0.74
	IJA-Dataset	93	100	96	96	1.79	91	52	66	73	1.03
	GTH-Dataset	96	100	98	98	0.91	95	55	70	76	0.87
ICDT ^[36]	D4J-Dataset	67	100	80	75	1.79	66	99	80	74	1.87
	IJA-Dataset	92	100	96	95	0.46	92	93	93	93	0.22
	GTH-Dataset	71	100	83	80	2.28	70	95	81	78	2.31
AYAT ^[37]	D4J-Dataset	85	110	92	91	2.83	88	96	92	92	2.81
	IJA-Dataset	82	100	90	89	4.17	84	89	86	86	4.32
	GTH-Dataset	81	100	90	89	1.50	80	83	81	81	1.45
Liu et al.'s technique ^[41]	D4J-Dataset	58	100	74	65	1595.18	59	100	74	66	1726.90
	IJA-Dataset	99	100	99	99	1388.05	99	100	99	99	13784.00
	GTH-Dataset	54	100	70	59	1405.00	54	100	70	59	1311.03
LWH ^[38]	D4J-Dataset	75	100	85	83	2.37	75	100	85	83	2.92
	IJA-Dataset	60	96	74	67	6.73	60	94	73	66	6.78
	GTH-Dataset	64	94	76	71	2.50	64	91	75	69	2.34

approach's performance, slightly less effective across other clone types.

Other techniques demonstrate varied efficacy. ICDT^[36] and AYAT^[37] achieve average precisions of 75%–77% and 73%–82%, respectively. VUDDY^[35] achieves a moderate average precision of 62%–63%. Further, Song et al.'s technique^[39] and Liu et al.'s technique^[41] achieve average precisions around 60%–65% and around 70%, respectively. Whereas Yang et al.'s technique^[40] and LWH^[38] achieve average precisions of 55%–56% and 50%–67%, respectively, which leads to miss potential clones. The comparison results reveal that while some techniques demonstrate strength in specific clone types, our technique maintains exceptional consistent precision across all clone types, emphasising its effectiveness in file-level clone detection.

5.5.2 Recall

In assessing file-level clone detection, our approach outperforms with an average recall of 99% across Type-I, Type-II, Type-III ST, and Type-III MT clones over all datasets, emphasising its ability to identify all types of clones. VCIPR^[34] and Liu et al.'s technique^[41] achieve impressive average recall rates of 100% in Type-I clones, demonstrating exceptional detection ability, though VCIPR's^[34] performance notably declines in Type-II, Type-III ST, and Type-III MT clones, showing different efficiency across clone types.

ICDT^[36] excels with a good average recall, securing 100% in Type-I and maintaining high effectiveness with 96%, 93%, and 89% in subsequent clone types, reflecting its ability to detect all clone types. AYAT^[37] outperforms, astonishingly achieving an average recall of 100% in Type-I, but its efficacy drops for Type-II,

Table 6 File-based direct clones evaluation metrics. D.Time stands for detection time.

Technique	Dataset	Type-III ST					Type-III MT				
		Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	D.Time (s)
Our technique	D4J-Dataset	100	100	100	100	0.91	100	100	100	100	0.95
	IJA-Dataset	98	100	99	99	2.03	98	100	99	99	2.10
	GTH-Dataset	96	97	96	96	0.85	96	96	96	96	0.82
VUDDY ^[35]	D4J-Dataset	67	91	77	73	0.25	66	91	77	72	0.26
	IJA-Dataset	60	75	67	62	0.33	59	74	66	62	0.38
	GTH-Dataset	59	73	65	61	0.22	60	72	65	61	0.23
Song et al.'s technique ^[39]	D4J-Dataset	66	93	77	72	4.16	65	93	77	71	6.27
	IJA-Dataset	60	80	69	64	7.17	60	78	68	62	7.73
	GTH-Dataset	60	80	69	64	7.85	61	80	69	65	5.15
Yang et al.'s technique ^[40]	D4J-Dataset	55	73	63	56	1.11	52	76	62	54	1.06
	IJA-Dataset	53	62	57	54	1.65	52	57	54	52	1.68
	GTH-Dataset	51	75	60	51	0.82	50	67	57	49	0.85
VCIPR ^[34]	D4J-Dataset	96	49	65	73	0.33	98	47	64	73	0.32
	IJA-Dataset	89	40	55	68	0.53	90	36	51	66	0.58
	GTH-Dataset	96	52	68	75	0.22	96	46	62	72	0.27
ICDT ^[36]	D4J-Dataset	67	98	80	75	1.96	64	98	78	72	2.05
	IJA-Dataset	93	87	90	90	0.33	89	74	81	82	0.35
	GTH-Dataset	71	94	81	78	2.21	71	94	81	78	1.99
AYAT ^[37]	D4J-Dataset	89	96	92	92	2.80	57	100	73	63	2.87
	IJA-Dataset	84	89	86	86	4.36	81	89	85	84	4.44
	GTH-Dataset	80	83	81	81	1.51	81	83	82	81	1.59
Liu et al.'s technique ^[41]	D4J-Dataset	59	100	74	66	1781.69	60	100	75	67	1791.41
	IJA-Dataset	99	99	99	99	15 113.00	98	97	97	50	19 058.00
	GTH-Dataset	54	100	70	59	1522.82	54	100	70	59	1788.91
LWH ^[38]	D4J-Dataset	53	100	69	55	0.71	52	100	68	53	0.83
	IJA-Dataset	50	92	65	51	2.55	50	92	65	51	3.68
	GTH-Dataset	53	98	69	55	4.46	51	98	67	52	4.32

Type-III ST, and Type-III MT clones to an average of 91%. Contrarily, VUDDY^[35] and Song et al.'s technique^[39] demonstrate average recall rates ranging in 88%–89% and 84%–98%, respectively, in different clone types. On the other hand, Yang et al.'s technique^[40] achieve average recall of 83%, 74%, 70%, and 67% for Type-I, Type-II, Type-III ST, and Type-III MT, respectively. Whereas LWH^[38] achieves near-perfect recall in the more complex clone types.

5.5.3 F1-score

In evaluating file-level clones, our approach outperforms with an average F1-score of 99% for Type-I and Type-II clones, and 98% for Type-III ST and Type-III MT clones, demonstrating its exceptional precision and recall balance across various clone types. VCIPR^[34] stands out with a commendable average of 96% in Type-I clones, yet its performance drops to

average F1-scores of 68%, 63%, and 59% in Type-II, Type-III ST, and Type-III MT clones, respectively, indicating a fluctuation in effectiveness across clone types.

ICDT^[36] and AYAT^[37] show remarkable proficiency, with ICDT^[36] achieving average F1-scores of 86%, 85%, 84%, and 80%, and AYAT^[37] reaching 90%, 81%, 81%, and 79% across the clone types, respectively. On the other hand, VUDDY^[35] demonstrates average scores of 74%, 70%, 70%, and 69%, reflecting a potential for improvement across clone types. Song et al.'s technique^[39], Liu et al.'s technique^[41], and LWH^[38] reveal their detection capabilities with Song et al.'s technique^[39] achieving average F1-scores of 76%, 72%, 72%, and 71%, Liu et al.^[41] achieving a consistent 81%, and LWH^[38] achieving 78%, 78%, 68%, and 67%. Whereas Yang

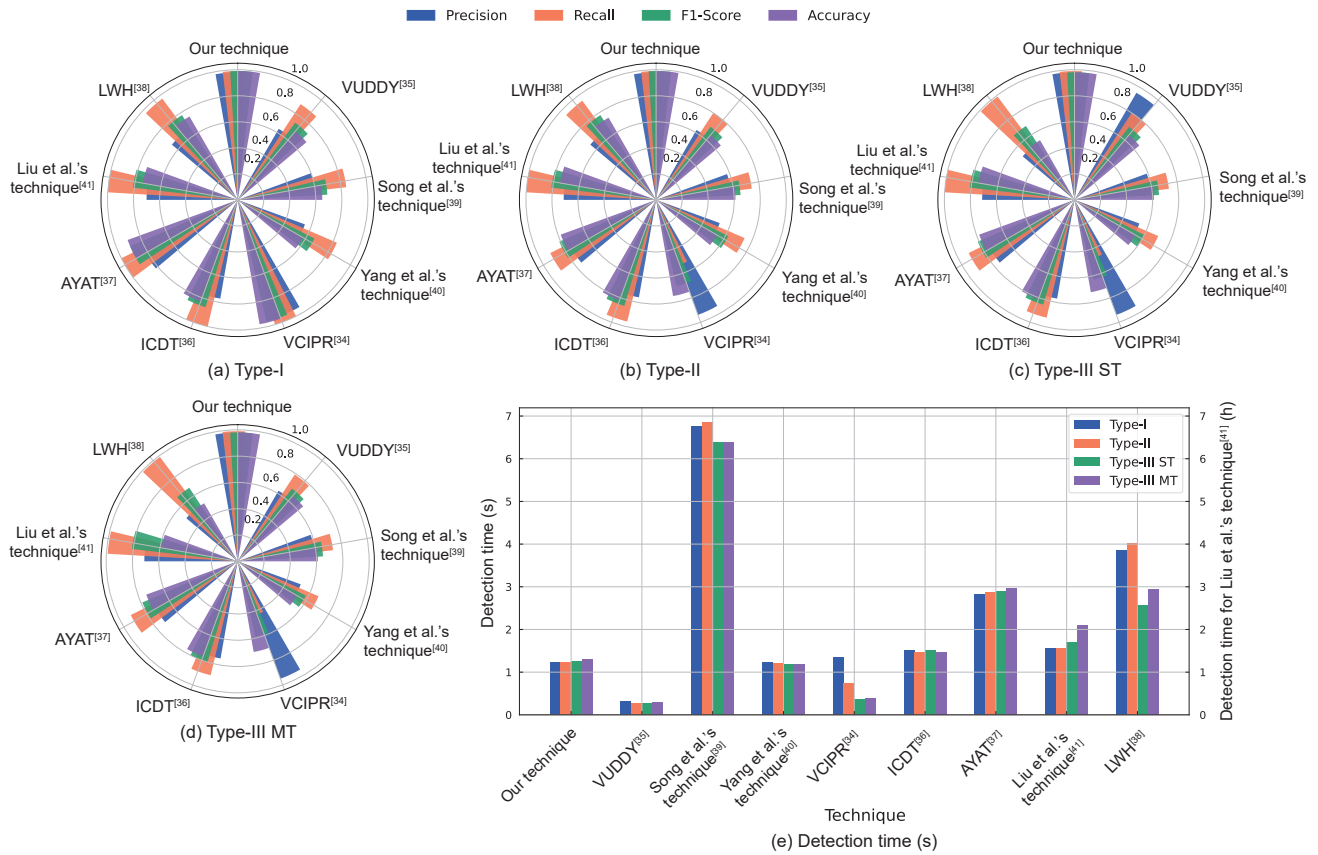


Fig. 7 Average file-based clone evaluation metrics over all datasets.

et al.'s technique^[40] achieves average F1-scores of 66%, 62%, 60%, and 58%.

5.5.4 Accuracy

Our approach maintains its dominance with an unmatched average accuracy of 99% for Type-I and Type-II clones, along with 98% for Type-III ST and Type-III MT clones, demonstrating its outstanding detection capabilities across all clone types. VCIPR^[34] follows closely, with an impressive average accuracy of 97% for Type-I clones, though its accuracy declines in performance for Type-II, Type-III ST, and Type-III MT clones with average accuracies of 75%, 72%, and 70%, respectively.

ICDT^[36] and AYAT^[37] demonstrate average proficiency, with ICDT^[36] achieving average accuracies of 83%, 82%, 81%, and 77%, and AYAT^[37] achieving 90%, 78%, 78%, and 74% across the Type-I–Type-III ST clones. VUDDY^[35], however, shows average accuracies of 69%, 65%, 65%, and 65% across the Type-I–Type-III ST clones. Song et al.'s technique^[39] and Yang et al.'s technique^[40] demonstrate their capabilities with Song et al.^[39] achieving an average accuracies of 71%, 67%, 67%,

and 67%, and Liu et al.'s technique^[41] demonstrating an average accuracy range of 58% to 75% for Type-I–Type-III ST, while its accuracy drops to 59% in Type-III MT clones. LWH^[38] and Yang et al.'s technique^[40] demonstrate their performance with LWH^[38] achieving average accuracies of 74%, 73%, 53%, and 51%, and Yang et al.'s technique^[40] slightly following with 58%, 54%, 54%, and 52% in the respective clone types.

5.5.5 Efficiency (detection time)

In the clone detection area, an effective technique must be efficient along with good precision, recall, F1-score, and accuracy. CloneTracker stands out, achieving outstanding detection times across various clone types over all datasets. From Type-I to Type-III MT clones, it detected clones in average times of 1.23, 1.24, 1.26, and 1.29 s respectively. VUDDY^[35], Song et al.'s technique^[39], and Yang et al.'s technique^[40] show varied efficiency. VUDDY's^[35] detection times range from 0.31 to 0.29 s for Type-I–Type-III MT clones, demonstrating excellent efficiency but struggling with accuracy, particularly for more complex clones. VCIPR^[34] offers moderate detection times but

struggles with precision, recall, F1-score, and accuracy. Our technique, by comparison, offers consistent performance, balancing speed with improved performance metrics across all clone types. Song et al.'s technique^[39] displays slower detection times, ranging from 6.75 to 6.38 s. Yang et al.'s technique^[40] offers balanced efficiency, with detection times between 1.19 and 1.25 s. ICDT^[36] and AYAT^[37] show stable performance, with ICDT^[36] and AYAT^[37] maintaining average speeds around 1.46 to 1.51 and 2.83 to 2.97 s, respectively, across all clone types. Whereas LWH^[38] reports 3.86, 4.01, 2.5, and 2.94 s, respectively, for Type-I, Type-II, Type-III ST, and Type-III MT clones, showing a balanced performance across all clone categories. ICDT^[36], AYAT^[37], LWH^[38] techniques take slightly longer to detect clones than CloneTracker.

Remarkably, in Liu et al.'s technique^[41], the use of the Smith–Waterman algorithm for matching significantly impacts its detection times, especially for large databases, where it extends beyond 5611.72, 5607.31, 6139.17, and 7546.10 s for Type-I, Type-II, Type-III ST, and Type-III MT clones. Smith–Waterman algorithm, while advantageous for accuracy, results in substantial computational demands, particularly when processing extensive databases. This method's complexity is primarily due to its complex matrix-based computations, which meticulously compare every possible segment of subject code against extensive databases to find the best alignment.

5.6 Database generation

In evaluating code clone detection techniques, the efficiency of database generation is also an essential metric, even though it is a one-time process. Our CloneTracker approach excels in key performance metrics such as precision, recall, F1-score, and accuracy, making it highly effective at detecting

clones. It takes 1980.3 s to generate the signature vault for D4J-Dataset, while its database generation time is slightly longer than some other techniques, such as AYAT's^[37] 746.3 s for the D4J-Dataset, it still significantly outperforms competitors such as VCIPR^[34], which takes 2869.4 s, as illustrated in Fig. 8, which provides a graphical comparison of the average database generation times across different techniques.

For IJA-Dataset, our approach maintains its efficiency with a generation time of 2578.3 s, significantly outperforming Yang et al.'s technique^[40], VCIPR^[34], and LWH^[38], which took 3313.24 s, 4274.05 s, and 2826.6 s, respectively. Meanwhile, VUDDY^[35] takes 1780.9 s, Song et al.'s technique^[39] takes 1611.5 s, ICDT^[36] takes 2010.5 s, and Liu et al.'s technique^[41] took 814.6 s to generate the database. For the GTH-Dataset, our approach has a generation time of 2144.5 s. VUDDY^[35] takes 1621.9 s, Song et al.'s technique^[39] takes 1298.15 s, Yang et al.'s technique^[40] takes 1303.2 s, VCIPR^[34] takes 1925.5 s, ICDT^[36] takes 1099.7 s, AYAT^[37] takes 2127.8 s, Liu et al.'s technique^[41] takes 752.1 s, and LWH^[38] takes 1383.2 s.

Although our approach is slightly slower in database generation due to its dual-layered nature and comprehensive preprocessing steps such as abstraction at each level, extracting functions, and generating chunks from abstracted code, this preprocessing enhances its precision, recall, F1-score, and accuracy in detection. Since database generation is a one-time process, the slightly longer duration can be ignored, given the advantages it brings to the overall performance of CloneTracker.

The system specification of this study is as below. A 64-bit Windows 11 operating system, equipped with an Intel i5 CPU 13500H v5 @2.6 GHz 2.4 GHz and

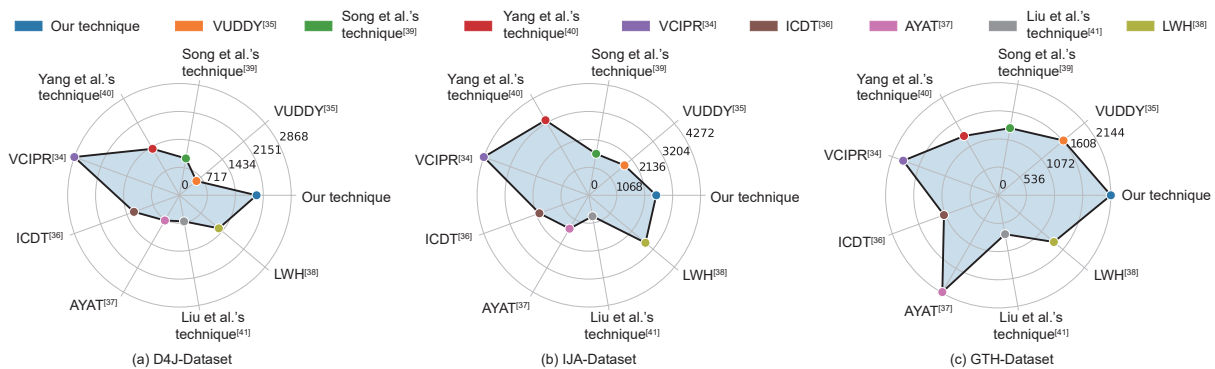


Fig. 8 Database generation time comparison on the (a) D4J, (b) IJA, and (c) GTH datasets.

16 GB LPDDR5 RAM, is utilized for generating the signature vault and detecting clones. Within this same setup, VUDDY^[35], Song et al.'s technique^[39], Yang et al.'s technique^[40], VCIPR^[34], ICDT^[36], AYAT^[37], Liu et al.'s technique^[41], and LWH^[38] are evaluated, with all employing batch processing for their assessments.

6 Discussion

A code cloning technique should be adopted in practice only if the proposed clone detection method proves effective by meeting evaluation metrics like high precision, recall, F1-score, and accuracy. This means the approach should ensure lower FP, FN, and higher TP and TN ratios, demonstrating both effectiveness and efficiency. If the technique takes longer to detect clones, that technique might not be suitable for a real-world scenario. Effectiveness and efficiency depend on each technique's preprocessing and processing phases.

CloneTracker distinguishes itself with advanced preprocessing and processing phases. It eliminates whitespaces and comments, which are fundamental for Type-I clone detection. This step improves the precision in identifying Type-I clones. The technique then applies advanced abstractions at various levels, including variables, data types, literals, functions, and classes, etc., to precisely detect Type-II clones. It prioritizes function-level detection, excluding function signatures for code under five tokens to avoid false positives, as smaller functions with abstractions will lead to higher false positive clones.

During its processing phase, CloneTracker employs an advanced matching method for clone identification, targeting a similarity range from 50% to 100% for both Type-III ST and Type-III MT. It starts with matching function signatures using an index and progresses to a refined index-based chunk-matching algorithm for cases that initially do not match (for Type-III ST and Type-III MT). This algorithm analyses chunks composed of six overlapping tokens, identifying them as clones if at least three chunks from the same file share at least 50% similarity, irrespective of their order and whether they are consecutive. Upon detecting clones, CloneTracker not only reveals the clone but also identifies modifications made in cloned code using the diff algorithm, providing detailed insights. The CloneTracker approach, through its advanced preprocessing and processing stages, secures high levels of precision, recall, F1-score, accuracy, and

efficiency. This makes CloneTracker an indispensable tool for software development and maintenance.

Song et al.'s technique^[39] processes source code structural patterns by extracting functions, segmenting them into slices, and converting these slices to signatures with the Fowler–Noll–Vo Variant 1a (FNV-1a) hash function. It uses a binary bit-vector for function comparison, identifying clones by matching these vectors. However, its binary matching lacks abstraction and flexibility, leading to potential false positives and negatives in large databases. This inflexibility may result in mismatches due to code variations in Type-III ST and Type-III MT clones, adversely impacting the technique's precision, recall, F1-score, and accuracy.

VCIPR^[34] technique employs a preprocessing strategy that replaces code identifiers with “id” labels, incrementing the “id” number by 1 for each identifier that appears in the code, except for Java keywords. This approach may lead to ambiguities and inconsistencies. This can lead to losing context, essential for accurately identifying clones. The technique also requires exactly matching chunk sizes of 10, making it challenging to detect clones with minor modifications, affecting detection performance metrics like precision and accuracy. While VCIPR^[34] technique works very well for simpler Type-I clones in file-based detection, its effectiveness significantly drops for more complex Type-III ST and Type-III MT clones due to their reliance on exact chunk matching and inability to handle variations, leading to lower detection effectiveness.

LWH^[38] technique uses a template-based method to identify Type-I and Type-II clones by converting complex code into generic labels and standardizing syntactical variations to spot direct, simple clones. While effective for simpler clones, this technique struggles with Type-III ST and Type-III MT clones, where the complexity of modifications exceeds the template's abstraction capacity. This results in lower performance evaluation metrics for complex clone types.

Yang et al.'s technique^[40] breaks down functions into abstract syntax trees with node types, including class elements, code blocks, statements, expressions, operators, keywords, and literals. These nodes are transformed into a sequence of characters, providing a compact code representation stored in a database for unique function patterns. However, due to the large

function count in extensive codebases and the limited character variety, the local sequence alignment method can cause many false positives and negatives, impacting accurate clone identification in large databases. This challenge leads to the average performance of the AST technique in function and file-based clone detection testing environment.

VUDDY's^[35] clone detection starts by extracting code functions and applying hierarchical abstraction on data types and function calls but not on string, numeric, and constructor literals. This selective abstraction can cause false positives by wrongly identifying small non-clone functions as clones due to its abstraction. VUDDY^[35] employs MD5 hashes for function signatures but encounters difficulties with modifications such as insertions or deletions of lines of code, common in Type-III ST and Type-III MT clones, resulting in different hash values. However, VUDDY^[35] can be highly efficient and effective in detecting Type-I and some Type-II clones.

The AYAT^[37] and ICDT^[36] techniques demonstrate strong performance in function-based and file-based clone detection due to the use of smaller chunk sizes, which significantly influence precision, recall, F1-score, and accuracy. Appropriate chunk size is essential; if it is too small, it may incorrectly flag non-clones as clones; if it is too large, the true clones might be missed; for instance, the recently proposed SQVDT^[45] technique underperformed across all datasets. Specifically, function-based tests failed to identify any true positives, resulting in a precision of 0%, recall, and F1-score. This poor performance is primarily due to SQVDT's reliance on exact chunk signature matches and its use of larger chunk sizes, with a window size of 15. Despite demonstrating excellent performance in Type-1 file-based clone detection only, SQVDT's limitations in function-based testing lead us to exclude its comparison in our case study analysis.

On the other hand, Liu et al.^[41] introduced an innovative technique for clone detection using sequence alignment, a method distinctly different from traditional code cloning techniques. This technique involves preprocessing to save MD5 signatures of each LOC in a database. During the detection phase, it compares the subject's signatures with those in the database, identifying a clone if at least 50% of the signatures are similar. However, Liu et al.^[41] employed

Smith–Waterman algorithm to match the subject's hash signature against all database signatures. The Smith–Waterman algorithm's core principle is based on constructing a complex matrix to compare sequences for the best possible systematic alignment. This process is especially critical when identifying similar sequences within large datasets. Due to its complexity and the detailed examination required at every step, the Smith–Waterman algorithm also requires more computational resources. This complexity is primarily why this technique requires hours to complete the detection process, unlike other methods that can detect clones in seconds.

CloneTracker has outperformed in both function-based and file-based case studies, consistently achieving high precision, recall, F1-score, accuracy, and efficiency across various clone types. While other techniques suffer from detecting clones, especially in function-based test studies, it goes beyond clone detection to accurately identify and locate the code modifications, providing in-depth insights essential for detailed clone analysis. The robust performance of CloneTracker emphasises its reliability as a key technique for modern software development and maintenance. Furthermore, CloneTracker's precise detection and comprehensive analysis contribute directly to improving software quality. By accurately identifying and locating clones and modifications, developers streamline maintenance tasks, reducing redundant or conflicting code. This minimizes potential bugs, inconsistencies, and technical debt, leading to more stable and reliable software systems. Moreover, its ability to handle complex clones enhances code reusability, enabling faster development cycles while maintaining high standards of quality. This comprehensive approach positions CloneTracker as an essential tool for maintaining and enhancing software quality, supporting long-term stability and maintainability in modern software projects.

7 Conclusion

In this research, we propose CloneTracker, a novel, dual-layered approach for identifying all types of code clones. Our approach uses function-based and chunk-based detection, capturing both function and file-level clones, in both modified and unmodified states. Our approach adeptly identifies commonly modified code using synergistic abstractions and innovative detection

methods, covering Type-I, Type-II, and Type-III (ST and MT) clones. Despite its dual-layered design, CloneTracker utilizes an intelligent retrieval method for the efficient identification of clones.

We conduct a comprehensive case study that compared our technique with eight recently proposed methods, including token-based, hybrid-token, PDG, AST, and hybrid-text based methods. The comprehensive comparison demonstrates our approach's superior metrics in precision, recall, F1-score, accuracy, and efficiency (detection time). With outstanding averages of around 98% in precision, 97%–100% in recall, 97%–99% in F1-score, and 97.5%–99% in accuracy across function-based and file-based assessments, our approach outperforms other techniques in both types of scenarios, achieving efficient average detection times of 1.14 to 1.29 s. AYAT^[37] ranks second, achieving average values of 75%–82% in precision, 82%–94% in recall, 77%–87% in F1-scores, 78%–87% in accuracy, but detection time lags behind 2.8 to 5.62 s. Whereas, Yang et al.'s technique^[40] performs poorly in the case study, achieving only 49%–50% in precision, 61%–71% in recall, 54%–59% in F1-score, and 48%–52% in accuracy, with detection time of 0.39 to 1.25 s. Furthermore, our approach identifies post-cloning modifications through the diff algorithm, significantly improving its applicability in software maintenance. Comprehensive assessment across multiple datasets confirms CloneTracker as an exceptionally effective and efficient means for detecting all clone types, emphasising its pivotal role in future software maintenance and development practices.

Acknowledgment

This research was financially supported by the Key Research Program of the Chinese Ministry of Science and Technology (No. 2022YFB3103903).

Declaration of competing interest

The authors have no competing interests to declare that are relevant to the content of this article.

References

- [1] C. J. Kapser and M. W. Godfrey, Supporting the analysis of clones in software systems, *J. Softw. Maint. Evol.: Res. Pract.*, vol. 18, no. 2, pp. 61–82, 2006.
- [2] S. Ducasse, M. Rieger, and S. Demeyer, A language independent approach for detecting duplicated code, in *Proc. IEEE Int. Conf. Software Maintenance-1999 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Oxford, UK, 1999, pp. 109–118.
- [3] M. Morshed, A. Rahman, and S. U. Ahmed, A literature review of code clone analysis to improve software maintenance process, arXiv preprint arXiv: 1205.5615, 2012.
- [4] S. Jadon, Code clones detection using machine learning technique: Support vector machine, in *Proc. 2016 Int. Conf. Computing, Communication and Automation*, Greater Noida, India, 2016, pp. 299–303.
- [5] Y. Nakamura, E. Choi, N. Yoshida, S. Haruna, and K. Inoue, Towards detection and analysis of interlanguage clones for multilingual web applications, in *Proc. 2016 IEEE 23rd Int. Conf. Software Analysis, Evolution, and Reengineering*, Osaka, Japan, 2016, pp. 17–18.
- [6] D. Yu, J. Wang, Q. Wu, J. Yang, J. Wang, W. Yang, and W. Yan, Detecting java code clones with multi-granularities based on bytecode, in *Proc. 2017 IEEE 41st Annu. Computer Software and Applications Conf.*, Turin, Italy, 2017, pp. 317–326.
- [7] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, Detecting android malware using clone detection, *J. Comput. Sci. Technol.*, vol. 30, no. 5, pp. 942–956, 2015.
- [8] F. Lyu, Y. Lin, J. Yang, and J. Zhou, SUIDroid: An efficient hardening-resilient approach to android app clone detection, in *Proc. 2016 IEEE Trustcom/BigDataSE/ISPA*, Tianjin, China, 2016, pp. 511–518.
- [9] M. A. Nishi and K. Damevski, Scalable code clone detection and search based on adaptive prefix filtering, *J. Syst. Softw.*, vol. 137, pp. 130–142, 2018.
- [10] Y. Glani, L. Ping, and S. A. Shah, AASH: A lightweight and efficient static IoT malware detection technique at source code level, in *Proc. 2022 3rd Asia Conf. Computers and Communications*, Shanghai, China, 2022, pp. 19–23.
- [11] Y. Glani, L. Ping, K. Lin, and S. A. Shah, AyatDroid: A lightweight code cloning technique using different static features, in *Proc. 2023 IEEE 3rd Int. Conf. Software Engineering and Artificial Intelligence*, Xiamen, China, 2023, pp. 17–21.
- [12] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, CCAAligner: A token based large-gap clone detector, in *Proc. 40th Int. Conf. Software Engineering*, Gothenburg, Sweden, 2018, pp. 1066–1077.
- [13] H. Sajjani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, SourcererCC: Scaling code clone detection to big-code, in *Proc. 38th Int. Conf. Software Engineering*, Austin, TX, USA, 2016, pp. 1157–1168.
- [14] Y. Yuki, Y. Higo, and S. Kusumoto, A technique to detect multi-grained code clones, in *Proc. 2017 IEEE 11th Int. Workshop on Software Clones*, Klagenfurt, Austria, 2017, pp. 1–7.
- [15] Y. Yang, Z. Ying, X. Hu, D. Lo and C. Ni, J. Grundy, X. Xia, C3: Code Clone-Based Identification of Duplicated

- Components, in *Proc. 31st ACM Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering*, San Francisco, CA, USA, 2023, pp. 1832–1843.
- [16] S. Chodarev, E. Pietriková, and J. Kollár, Haskell clone detection using pattern comparing algorithm, in *Proc. 2015 13th Int. Conf. Engineering of Modern Electric Systems*, Oradea, Romania, 2015, pp. 1–4.
- [17] J. Pati, B. Kumar, D. Manjhi, and K. K. Shukla, A comparison among ARIMA, BP-NN, and MOGA-NN for software clone evolution prediction, *IEEE Access*, vol. 5, pp. 11841–11851, 2017.
- [18] S. Sargsyan, S. Kurmangaleev, A. Belevantsev, and A. Avetisyan, Scalable and accurate detection of code clones, *Program. Comput. Softw.*, vol. 42, no. 1, pp. 27–33, 2016.
- [19] C. M. Kamalpriya and P. Singh, Enhancing program dependency graph based clone detection using approximate subgraph matching, in *Proc. 2017 IEEE 11th Int. Workshop on Software Clones*, Klagenfurt, Austria, 2017, pp. 1–7.
- [20] M. Wang, P. Wang, and Y. Xu, CCSharp: An efficient three-phase code clone detector using modified PDGs, in *Proc. 2017 24th Asia-Pacific Software Engineering Conf.*, Nanjing, China, 2017, pp. 100–109.
- [21] Y. Sabi, Y. Higo, and S. Kusumoto, Rearranging the order of program statements for code clone detection, in *Proc. 2017 IEEE 11th Int. Workshop on Software Clones*, Klagenfurt, Austria, 2017, pp. 1–7.
- [22] J. Crussell, C. Gibler, and H. Chen, AnDarwin: Scalable detection of android application clones based on semantics, *IEEE Trans. Mob. Comput.*, vol. 14, no. 10, pp. 2007–2019, 2015.
- [23] T. Vislavski, G. Rakić, N. Cardozo, and Z. Budimac, LICCA: A tool for cross-language clone detection, in *Proc. 2018 IEEE 25th Int. Conf. Software Analysis, Evolution and Reengineering*, Campobasso, Italy, 2018, pp. 512–516.
- [24] R. Tekchandani, R. Bhatia, and M. Singh, Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis, *J. Supercomput.*, vol. 74, no. 9, pp. 4199–4226, 2018.
- [25] Roopam and G. Singh, To enhance the code clone detection algorithm by using hybrid approach for detection of code clones, in *Proc. 2017 Int. Conf. Intelligent Computing and Control Systems*, Madurai, India, 2017, pp. 192–198.
- [26] R. H. Misu and K. Sakib, Interface driven code clone detection, in *Proc. 2017 24th Asia-Pacific Software Engineering Conf.*, Nanjing, China, 2017, pp. 747–748.
- [27] T. Matsushita and I. Sasano, Detecting code clones with gaps by function applications, in *Proc. 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, Paris, France, 2017, pp. 12–22.
- [28] K. Uemura, A. Mori, K. Fujiwara, E. Choi, and H. Iida, Detecting and analyzing code clones in HDL, in *Proc. 2017 IEEE 11th Int. Workshop on Software Clones*, Klagenfurt, Austria, 2017, pp. 1–7.
- [29] N. Saini, S. Singh, and Suman, Code clones: Detection and management, *Procedia Comput. Sci.*, vol. 132, pp. 718–727, 2018.
- [30] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, On software maintenance process improvement based on code clone analysis, in *Proc. 4th Int. Conf. Product Focused Software Process Improvement*, Rovaniemi, Finland, 2002, pp. 185–197.
- [31] D. Li, M. Piao, H. S. Shon, K. H. Ryu, and I. Paik, One pass preprocessing for token-based source code clone detection, in *Proc. 2014 IEEE 6th Int. Conf. Awareness Science and Technology*, Paris, France, 2014, pp. 1–6.
- [32] M. Westerdahl, <https://github.com/JCash>, 2024
- [33] Hash Benchmark, <https://xxhash.com/>, 2024
- [34] J. Akram, L. Qi, and P. Luo, VCIPR: Vulnerable code is identifiable when a patch is released (hacker’s perspective), in *Proc. 2019 12th IEEE Conf. Software Testing, Validation and Verification*, Xi’an, China, 2019, pp. 402–413.
- [35] S. Kim, S. Woo, H. Lee, and H. Oh, VUDDY: A scalable approach for vulnerable code clone discovery, in *Proc. 2017 IEEE Symp. Security and Privacy*, San Jose, CA, USA, 2017, pp. 595–614.
- [36] A. A. Elkhail, J. Svacina, and T. Cerny, Intelligent token-based code clone detection system for large scale source code, in *Proc. Conf. Research in Adaptive and Convergent Systems*, Chongqing, China, 2019, pp. 256–260.
- [37] Y. Giani, L. Ping, and S. A. Shah, AYAT: A lightweight and efficient code clone detection technique, in *Proc. 2022 3rd Asia Conf. Computers and Communications*, Shanghai, China, 2022, pp. 47–52.
- [38] E. Kodhai and S. Kanmani, Method-level code clone detection through LWH (Light Weight Hybrid) approach, *J. Softw. Eng. Res. Dev.*, vol. 2, no. 1, p. 12, 2014.
- [39] X. Song, A. Yu, H. Yu, S. Liu, X. Bai, L. Cai, and D. Meng, Program slice based vulnerable code clone detection, in *Proc. 2020 IEEE 19th Int. Conf. Trust, Security and Privacy in Computing and Communications*, Guangzhou, China, 2020, pp. 293–300.
- [40] Y. Yang, Z. Ren, X. Chen, and H. Jiang, Structural function based code clone detection using a new hybrid technique, in *Proc. 2018 IEEE 42nd Annu. Computer Software and Applications Conf.*, Tokyo, Japan, 2018, pp. 286–291.
- [41] J. Liu, T. Wang, C. Feng, H. Wang, and D. Li, A large-gap clone detection approach using sequence alignment via dynamic parameter optimization, *IEEE Access*, vol. 7, pp. 131270–131281, 2019.
- [42] J. Svajlenko and C. K. Roy, Evaluating clone detection tools with BigCloneBench, in *Proc. 2015 IEEE Int. Conf. Software Maintenance and Evolution*, Bremen, Germany, 2015, pp. 131–140.

- [43] J. Svajlenko and C. K. Roy, BigCloneEval: A clone detection tool evaluation framework with BigCloneBench, in *Proc. 2016 IEEE Int. Conf. Software Maintenance and Evolution*, Raleigh, NC, USA, 2016, pp. 596–600.
- [44] BigCloneBench, <https://github.com/clonebench/BigCloneBench>, 2024.
- [45] J. Akram and P. Luo, SQVDT: A scalable quantitative vulnerability detection technique for source code security assessment, *Softw.: Pract. Exper.*, vol. 51, no. 2, pp. 294–318, 2021.
- [46] Github Java Corpus. <https://groups.inf.ed.ac.uk/cup/javaGithub/>, Accessed: February 2024.
- [47] Data for Java. <https://www.kaggle.com/datasets/zavadsky/lots-of-code>, 2024.



Yasir Glani received the bachelor degree in software engineering from Nanjing University of Aeronautics and Astronautics, Nanjing, China, and the master degree in information security from Shanghai Jiao Tong University, Shanghai, China, and the PhD degree in software engineering from Tsinghua University, Beijing, China, in 2024. He is currently a postdoctoral researcher at School of Software, Northwestern Polytechnical University, Xi'an, China. His research interests include code cloning and software maintenance. Prior to this, he served as a senior software testing engineer at Huawei Technology Co., Ltd., Shanghai, China, focusing on 5G telecommunication technology.



analysis, code cloning, and database security.

Ping Luo received the PhD degree in applied mathematics from the Chinese Academy of Sciences Institute of Systems Science, Beijing, China, in 1996. He is currently a professor and a PhD supervisor at Tsinghua University, Beijing, China. His research interests include cryptographic algorithms, vulnerability



in quantum computing, his research interests include the quantum Petri net model and developing quantum algorithms.

Syed Asad Shah received the master degree in cyber space security from Shanghai Jiao Tong University, Shanghai, China, in 2020, and the bachelor degree in computer science from University of the Punjab, Lahore, Pakistan, in 2016. He is currently a PhD candidate at Bilkent University, Ankara, Türkiye. Specializing



computation, natural language processing, and steganography.

Ke Lin received the master degree in software engineering from Tsinghua University, Beijing, China, in 2025, and the bachelor degree in software engineering from Tsinghua University, Beijing, China, in 2022. He is currently a PhD candidate in data science at College of William & Mary, Williamsburg, VA,